AD-A045 593   CALIFORNIA UNIV SANTA CRUZ INFORMATION SCIENCES   F/G 9/4
              A LINGUISTIC APPROACH TO AUTOMATIC THEOREM PROVING.(U)
              AUG 76 S SICKEL   N00014-76-C-0681
JNCLASSIFIED   NL

| OF |
AD
A045593

END
DATE
FILMED
11 - 77
DDC

A LINGUISTIC APPROACH TO
AUTOMATIC THEOREM PROVING

by

SHARON SICKEL

APPEARED IN

CSCSI/SCEIO SUMMER CONFERENCE 1976

PROCEEDINGS

UNIVERSITY OF CALIFORNIA
INFORMATION SCIENCES
SANTA CRUZ, CA. 95064

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS
BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)	5. TYPE OF REPORT & PERIOD COVERED
A LINGUISTIC APPROACH TO AUTOMATIC THEOREM PROVING.	Technical rept.,
	6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)	8. CONTRACT OR GRANT NUMBER(s)
Sharon/Sickel	NØØØ14-76-C-Ø681

9. PERFORMING ORGANIZATION NAME AND ADDRESS	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
Information Sciences University of California Santa Cruz, California 95064	

11. CONTROLLING OFFICE NAME AND ADDRESS	12. REPORT DATE
Office of Naval Research Arlington, Virginia 22217	August 1976
	13. NUMBER OF PAGES
	10

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report)
Office of Naval Research University of California 553 Evans Hall Berkeley, California 94720	Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited. It may be released to the Clearinghouse, Department of Commerce, for sale to the general public.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

D D C
RECEIVED
OCT 26 1977
C

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Automatic theorem proving, formal grammars, regular expressions, integer programming

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A method is discussed that maps theorem proving using clause interconnectivity graphs onto formal grammars. The languages generated by the grammars relate to the proofs of the theorems.

A LINGUISTIC APPROACH TO AUTOMATIC THEOREM PROVING

Sharon Sickel
Information Sciences
University of California, Santa Cruz, Ca.

ABSTRACT

Generalizing the concept of a path in Clause Interconnectivity Graphs, we define the set of simple (i.e., cycle-free) paths that begin at a specified subset of nodes. Where the search of the CIG for a proof in the predicate calculus was previously defined in terms of the edges of the CIG, here the simple paths themselves become the atomic elements of the search, thereby increasing the "chunk" size of the operands. We can further define forms similar to regular expressions in which the terminal symbols represent those simple chunks. The forms become templates that model proofs, i.e., they can be mapped onto resolution proofs of the unsatisfiability of the clauses making up the CIG. In general a template represents an infinite number of paths but an algebraic computation on information derived from the t___ es yields valid proofs without an exhaustive search through intermediate ____ of the search tree. Overall, the method leads to a reduction in both the computation time per step as well as in the combinatorics of the search itself. The representation also lends itself to an heuristic based on integer programming by using a simple difference function based on the chunks.

Introduction

A system for formal theorem proving is presented, using the Clause Interconnectivity Graph as its basic data structure. Proofs found here can be mapped onto proofs using resolution and factoring as rules of inference (as opposed to Modus Ponens, for example). The search method bears little resemblance to that of resolution methods, however.

The Clause Interconnectivity Graph (CIG) [5] has been used as a representation for proving first-order predicate calculus theorems. A CIG is a four-tuple:

< Nodes, Edges, Subst, Clause > where

1

Linguistic Approach...

Nodes is a set of graph nodes, one for each literal of each clause,

Edges is a symmetric relation between nodes such that $\langle a,b \rangle \in$

Edges iff the literals associated with nodes a and b have

opposite signs and unifiable atoms.

Subst is a mapping: Edges → substitutions such that

Subst($\langle a,b \rangle$) is a most general unifier of the atoms of the

literals associated with nodes a and b, and

Clause is a mapping: Nodes → $\mathcal{P}$(Nodes) where $\mathcal{P}$ means powerset;

Clause partitions the nodes so that literals in the same

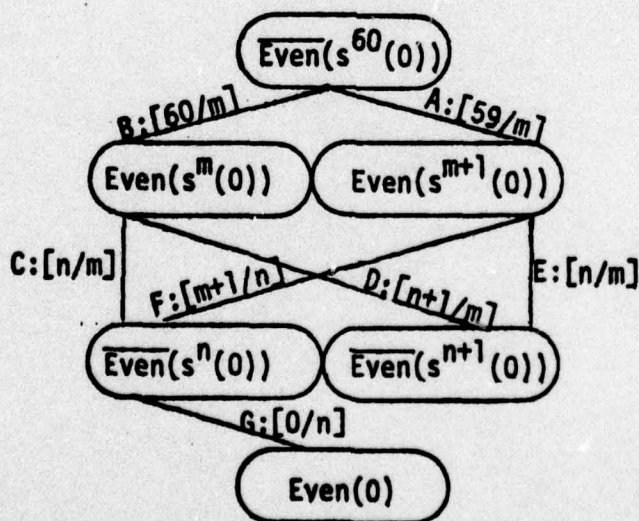clause have corresponding nodes in the same partition.

For example, suppose that we are dealing with integers defined by Peano's

axioms, and we define the predicate, Even:

$Even(0)$

$Even(s^n(0)) \rightarrow \overline{Even}(s^{n+1}(0))$ [†]

$\overline{Even}(s^n(0)) \rightarrow Even(s^{n+1}(0))$

and theorem $Even(s^{60}(0))$. Then the CIG is shown in Figure 1.

Figure 1. A Clause Interconnectivity Graph with labeled edges. The predicates and terms are left in the nodes for expository purposes only. They are neither included in the CIG definition nor are they used in the search for a proof.

_____

[†] "s" means "successor"; $s(0) = 0$; $s^n(0) = s(s^{n-1}(0))$ for $n > 0$.

2

Linguistic Approach...

Edges is a symmetric relation. However, when we involve an edge in the search, the analogy is made to moving from one element of an ordered pair in Edges to the other element in that pair. Therefore when an edge is used, we think of it as being directed. Given an edge <a,b> and assuming direction $a \rightarrow b$, we can make the following definitions.

Deleting_literal is a mapping: Edges $\rightarrow$ Nodes where

Deleting_literal(<a,b>) = b and

Residual_literals is a mapping: Nodes $\rightarrow \mathcal{P}$(Nodes) where $\mathcal{P}$ means powerset.

Residual_literals(b) = Clause(b) - {b}.

A proof derived from a CIG corresponds to a particular kind of search on the CIG. The proof search resembles the following process:

Choose a clause to be the starting clause (a clause that is likely to be used in the proof, a member of the set of support, etc.). Place a marker on each of the nodes in the partition representing the starting clause. Each of those markers may be moved along any edge connected to its present position. Then the parent marker is removed (from the deleting node) and children markers are placed on each of the other nodes (the residual nodes) in the partition arrived at from the move. Then the process is repeated on all of the existing markers; they in turn become parents, being replaced by children. The goal is to eliminate all markers.[†] This process corresponds to unrolling the graph into trees.

From looking at the CIG in Figure 1, it is easy to see that some move sequences could be done an arbitrary number of times, e.g., moves D,F,D,F,... successively, or E,C,E,C,... We call such sequences loops.

Assuming starting clause Even(0), the first move is determined, namely G. That leaves a marker on the node corresponding to $\overline{Even}(s^{n+1}(0))$. From this node we could begin one of the loops mentioned above. Let us consider a sequence of moves involving one of the loops; $G(DF)^k DA$, meaning move along G, then around D and F k times, then along D, then A. Intuitively G links up the integer

---

† This process is over-simplified. There are restrictions concerning the substitutions, and there is another allowable move that admits non-input steps. For a complete description, see [5].

3

0 with the start of an induction. The DF loop adds the value 2 to the current value. Move A jumps out of the induction to the value that we seek. In other words, the $G(DF)^k$ part is successively proving that 0 is even, 2 is even, 4 is even, etc., until we arrive just short of the given value. The D and A steps together add 2 to the value. In this case, k will have the value 29.

Once we have discovered $G(DF)^k DA$, proofs of the evenness of all even, positive integers should be equally easy in all systems. But we know that they are not. Using traditional deductive systems on this axiomatization, the length of the proof of $Even(s^n(0))$ increases linearly with n, and required resources generally increase exponentially with the length of the proof. In this method, however, the discovery of the proof is of the same inherent difficulty regardless of the magnitude of n. The approach involves:

1) mapping the CIG onto a context-free grammar [1]

2) mapping the context-free grammar onto a set of expressions similar to regular expressions.

3) mapping each regular expression onto a composition of substitutions.

4) checking to see if any of the expressions represent a legal substitution.

If so, that expression can be mapped onto a proof.

Chunking

The previously presented search schemes on CIG's dealt with looping by preferring non-loop moves, preventing run-away development of infinite loops. However, even in some simple cases, we may need to travel a loop many times. One example of this is the proof of evenness in which we should be able to prove Even(6000) easily once the general method is discovered. The proof itself may be long, but the search time should be identical to the search time in proving Even(60) or Even(6). In fact, it is possible to use this method not only to prove individual theorems, but also to derive generalized algorithms to do computations within a theory.

Once we know the basic steps needed for a proof, the repetition of one or

more of those steps a large number of times should not cause us any trouble.
We need to discover these basic steps or chunks. One might imagine that the
moves that correspond to edges might serve satisfactorily as chunks. However,
there is some obvious clumping that takes place. The CIG in Figure 2 has three
natural chunks, C_1:f, C_2:deg, C_3:abc, because the moves within each chunk must
be taken together. Note that C_3 denotes a loop, and we can travel in either
direction on a loop, so we can denote cba as C_3^{-1}. In this case, the chunking
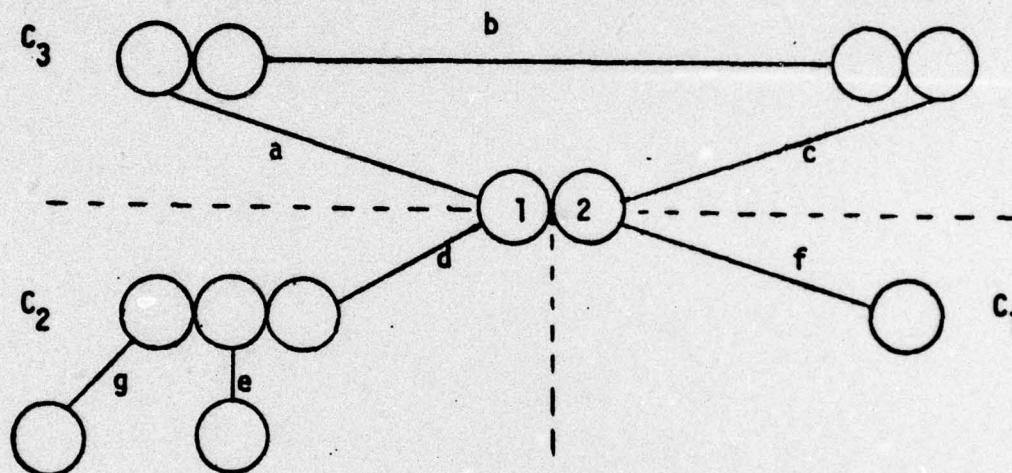partitioned the edges, but that will not necessarily be the case.

Figure 2. A CIG divided into its three natural chunks.

We can derive the chunks by finding all ways of moving and replacing the
markers such that if a marker is on the same node as one of its ancestors, we
freeze that marker, but continue to move other available markers. The starting
configuration for each chunk is a single marker sitting on some node. The
chunk is said to be related to that node. Intuitively, the chunk represents
the refutation of the literal that the related node represents. This process
identifies all of the natural pieces of the graph. Since no repeated looping
is allowed, this is a terminating process.

We classify the chunks into two types, terminal and loop. A <u>terminal</u>
chunk is one in which all markers have been eliminated. A <u>loop</u> chunk has one
or more frozen markers. In Figure 2, C_1 and C_2 are terminal chunks; C_3 is a

5

Linguistic Approach...

loop chunk.

Chunks to Context-Free Grammar

Chunks as described in the previous section are trees, since 1) a parent marker may be replaced by one or more children markers and 2) no marker can ever be its own descendant. We wish to write the chunks as linear sequences so that we can use them in constructing a grammar. We produce this flattening by doing an end-order traversal [4] of the "chunk tree". The flattened form is a sequence of directed edges and nodes, s_1, s_2, ... s_n. We can make context-free productions by putting s_1, s_2, ... s_n on the right-hand-side and the associated node on the left-hand-side,

$$N \rightarrow s_1 s_2 ... s_n.$$

The intuitive notion is that to eliminate N you must add $s_1, s_2, ... s_n$ (possibly including N). We can now construct a context-free grammar G:

nonterminals: {S} U Nodes   (where {S} ∩ Nodes = φ)

terminals: Edges

productions: {all $N \rightarrow s_1 s_2 ... s_n$ as described above}

U {S → $N_1 ... N_k$ | $N_1, ..., N_k$ represent all literals in
starting clause}

start symbol: S

In the ground case any string in the language of G, i.e. any string that is derivable from S and consists entirely of terminals (in this case edges), represents a proof. Therefore, once the chunking is accomplished, determining theoremhood of the statement in question is equivalent to asking whether a given context-free grammar generates a non-empty language, which is a trivial problem.

The general case is more difficult, however. Each edge has an associated substitution, and for a string of edges to be acceptable, all of their substitutions must be mutually consistent. Consistent($\alpha_1, \alpha_2, ... \alpha_n$) iff $\alpha_1 \odot (\alpha_2 \odot (... \odot \alpha_n)))$ is defined, where $\alpha \odot \beta = \gamma$ such that γ is a most

6

general substitution satisfying $(L\alpha)\gamma = (L\gamma)\alpha = L\gamma = (L\beta)\gamma = (L\gamma)\beta$ for an

arbitrary literal L [5]. Since all terminal strings must abide by consistency,

this is in fact a context-free attribute grammar [3] and can have the power of

a type 0 grammar. This fact eliminates the usefulness of the result that tells

us there is an upper bound on the length of the shortest string in the language.

However, the grammar form provides us with some valuable heuristics as we shall

see later.

Regular-like Expressions

Given a context-free grammar, it would be convenient to represent the

language generated in regular expression style. To do that, we need to extend

the definition of regular expression. In addition to "|", meaning "or",

concatenation meaning "and", and "*" meaning "repeat zero or more times", we

add exponent "n" mean repeat exactly n times.[†] For the grammar constructed

in the previ ... tion, if all productions that have node N on the left-hand-

side have one of $t_1, \ldots t_n$ (terminal chunks), or $r_1 N, \ldots r_k N$ (loop chunks), then,

intuitively, the expression $(r_1 | r_2 | \ldots r_k)^* t_1 | \ldots | t_n$ represents the refutation

for N and we denote it

$$N \overset{*}{\Rightarrow}{}^{\ddagger} (r_1 | r_2 | \ldots | r_k)^* (t_1 | \ldots | t_n).$$

I.e. we can go around loops as long and in whatever order we choose, but we

must finally end with a terminal.

In the example in Figure 2,

①$\overset{*}{\Rightarrow}$ $(abc)^* deg$, ②$\overset{*}{\Rightarrow}$ $(cba)^* f$.

It may be that by the above recursion method and by simple back-substitution

for nonterminals of right-hand-sides having the corresponding nonterminals on

the left, we can derive $S \overset{*}{\Rightarrow} p_1 p_2 \ldots p_n$ where $p_i \in$ Edges. For the example of

Figure 2, the grammar is:

---

† This notation appears frequently in the literature on formal languages.

‡ $A \overset{*}{\Rightarrow} B$ means B can be derived from A by an application of zero or more
   productions.

Linguistic Approach...

$((S, ①, ②...^\S), \{a,b,c,d,e,f,g\}, P, S)$ where P:

$S \to ①②$

$① \to a\ b\ c\ ①$

$① \to d\ e\ g$

$② \to c\ b\ a\ ②$

$② \to f$

By back-substitution we get: $S \overset{*}{\Rightarrow} (abc)^* deg(cba)^* f$. Now by replacing each terminal by its substitution and interpreting concatenation of substitutions to mean $\odot$, we can easily determine whether there exist non-negative integers n and m such that $subst^n(abc) \odot subst(def) \odot subst^m(cba) \odot subst(f)$ is defined. Note that we have replaced whole chunks by their substitutions. The substitution of a chunk is the $\odot$ composition of the substitutions of the edges making up the chunk. Each time a loop is repeated a new instance of the clause at the endpoints of the loop is added. For this reason, a loop repeated n times will have n __distinct__ instances of the variables. Loop substitutions, then, must be abstract descriptions including an unknown number of instances of variables. For example the substitution $[f(x_n)/x_{n+1}]$ specifies that each new instance of x is replaced by function "f" applied to the term substituted for the last instance of x.

For example, the grammar built from the CIG in Figure 1 having Even(0) as the start clause would cause S to generate (among others) the expression $G(DF)^* DA$. The corresponding substitution θ is

$$[0/n] \odot [n+1/m , m+1/n]^* \odot [n+1/m] \odot [59/m].$$

$$\left.\begin{array}{l} m_i = n_i + 1 \\ n_{i+1} = m_i + 1 \\ (1 \le i) \end{array}\right\} \Rightarrow \begin{array}{l} m_i = 2i+1 \\ n_i = 2i \\ (1 \le i) \end{array}$$

---

§ The other nonterminal names and their productions are irrelevant to this discussion.

8

Linguistic Approach...

Differentiating between instances of variables, θ becomes $[0/n_0] \odot [2i/n_1,$
$2i-1/m_{i-1}] \odot [n_k+1/m_k] \odot [59/m_k]$ where $1 \le i \le k$. $m_k = 59 = n_k+1 = 2k+1$,
therefore $k = 29$, indicating that the refutation consists of G, twenty-
nine repetitions of (CF) and finally D and A. We will not go into how to
generally describe loop substitutions, decide which instances of a variable
are referred to by other substitutions, or compute the exponent of loops.
However, for a given expression that is a regular expression extended by
exponents and contains no node names (i.e., is completely terminal), it is
straightforward to answer those questions. Due to lack of space the algo-
rithms will be presented in a subsequent paper.

Integer Programming Heuristic

There will be grammars derivable from CIG's that do not easily admit the
extended regular expressions. They include 1) grammars in which the self-
referencing non-terminal appears in the middle of the right-hand-side (e.g.,
$N \rightarrow aNb$) and 2) grammars in which a nonterminal can generate a string con-
taining two copies of itself, e.g., $N \overset{*}{=}> \alpha NN\beta$ where α and β are possibly
empty strings of symbols, i.e., $\alpha, \beta \in (\text{Edges} \cup \text{Nodes})^*$. In the latter case,
it is difficult to see the general recursion pattern since the length of the
resulting string is exponential with the number of repetitions. In both
cases keeping track of which instances of the variables to put in each sub-
stitution is a horrendous job in general.

By weakening the grammar, allowed by its particular use in this application,
and not by distinguishing between different instances of the same variable, we
can always derive an extended regular expression reduced to terminals, the
terminals possibly reordered from what the grammar would actually generate.

Every chunk has a (possibly empty) effect on the total substitution in a
solution. Terminal chunks have a fixed effect. Loop pieces may have a recur-
sive effect. E.g., $[f(x_k)/x_{k+1}]$ has the effect of adding f to the accumulated
effect and applying it to the new "x".

9

By combining the information from the reordered extended regular expression and the chunk effects, it is possible to write integer programming problems[2] whose solutions are likely candidates for proofs. In this way, the effects serve as difference functions for the chunks (operators) in much the way as is done in an operator difference table. The integer program tells us how many applications of each operator there are in likely candidates. The structure of the original grammar can then be used to check the validity of that candidate. An example of this is the "Even" problem in which we need to change the term from "0" in the start state to "$s^{60}(0)$" in the goal state. Therefore the sum of the effects of the chunks used must sum to exactly sixty applications of "s". In some cases, the start and goal states are not so clearly known and we have to phrase the problem slightly differently such that the original terms used in the solution plus the effects of all applied chunks sum to zero.

In cases where the regular expression forms are exactly known, the integer programming heuristic is substantially improved because the proper placement of variable instances is known. We may then break the problem into subproblems - one for each variable.

Work on the integer programming heuristic and computation of effects of more complex loops is currently in progress.

References

1. Hopcroft, John, and Jeffrey Ullman. Formal Languages and Their Relation to Automata. Addison Wesley, Menlo Park, CA. (1969)

2. Hu, T.C., Integer Programming and Network Flows. Addison Wesley, Menlo Park, CA. (1969).

3. Knuth, D. E., Semantics of Context-free Languages, Mathematical Systems Theory, 2 (Feb. 1968).

4. Knuth, D. E., The Art of Computer Programming, Vol 1. Addison-Wesley Menlo Park, CA (1969).

5. Sickel, Sharon, A Search Technique for Clause Interconnectivity Graphs, IEEE Transactions on Computers, Special Issue on Automatic Theorem Proving, (Aug. 1976).